Middleware for Incremental Processing in Conversational Agents

David Schlangen[*], **Timo Baumann**[*], **Hendrik Buschmeier**[†], **Okko Buß**[*]
Stefan Kopp[†], **Gabriel Skantze**[‡], **Ramin Yaghoubzadeh**[†]
[*]University of Potsdam Germany
[†]Bielefeld University Germany
[‡]KTH, Stockholm Sweden
david.schlangen@uni-potsdam.de

Abstract

We describe work done at three sites on designing conversational agents capable of incremental processing. We focus on the 'middleware' layer in these systems, which takes care of passing around and maintaining incremental information between the modules of such agents. All implementations are based on the abstract model of incremental dialogue processing proposed by Schlangen and Skantze (2009), and the paper shows what different instantiations of the model can look like given specific requirements and application areas.

1 Introduction

Schlangen and Skantze (2009) recently proposed an abstract model of incremental dialogue processing. While this model introduces useful concepts (briefly reviewed in the next section), it does not talk about how to actually implement such systems. We report here work done at three different sites on setting up conversational agents capable of incremental processing, inspired by the abstract model. More specifically, we discuss what may be called the 'middleware' layer in such systems, which takes care of passing around and maintaining incremental information between the modules of such agents. The three approaches illustrate a range of choices available in the implementation of such a middle layer. We will make our software available as development kits in the hope of fostering further research on incremental systems.[1]

In the next section, we briefly review the abstract model. We then describe the implementations created at Uni Bielefeld (BF), KTH Stockholm (KTH) and Uni Potsdam (UP). We close with a brief discussion of similarities and differences, and an outlook on further work.

[1]Links to the three packages described here can be found at http://purl.org/net/Middlewares-SIGdial2010.

2 The IU-Model of Incremental Processing

Schlangen and Skantze (2009) model incremental systems as consisting of a network of processing *modules*. Each module has a *left buffer*, a *processor*, and a *right buffer*, where the normal mode of processing is to take input from the left buffer, process it, and provide output in the right buffer, from where it goes to the next module's left buffer. (Top-down, expectation-based processing would work in the opposite direction.) Modules exchange *incremental units* (IUs), which are the smallest 'chunks' of information that can trigger connected modules into action. IUs typically are part of larger units; e.g., individual words as parts of an utterance, or frame elements as part of the representation of an utterance meaning. This relation of being part of the same larger unit is recorded through *same level links*; the information that was used in creating a given IU is linked to it via *grounded in* links. Modules have to be able to react to three basic situations: that IUs are *added* to a buffer, which triggers processing; that IUs that were erroneously hypothesised by an earlier module are *revoked*, which may trigger a revision of a module's own output; and that modules signal that they *commit* to an IU, that is, won't revoke it anymore (or, respectively, expect it to not be revoked anymore).

Implementations of this model then have to realise the actual details of this information flow, and must make available the basic module operations.

3 Sociable Agents Architecture

BF's implementation is based on the 'D-Bus' message bus system (Pennington et al., 2007), which is used for remote procedure calls and the bi-directional synchronisation of IUs, either locally between processes or over the network. The bus system provides *proxies*, which make the interface of a local object accessible remotely without copying data, thus ensuring that any access is guaranteed to yield up-to-date information. D-Bus bindings exist for most major programming languages, allowing

Proceedings of SIGDIAL 2010: the 11th Annual Meeting of the Special Interest Group on Discourse and Dialogue, pages 51–54,
The University of Tokyo, September 24-25, 2010.

for interoperability across various systems.

IUs exist as objects implementing a D-Bus interface, and are made available to other modules by publishing them on the bus. Modules are objects comprising a main thread and right and left buffers for holding own IUs and foreign IU proxies, respectively. Modules can co-exist in one process as threads or occupy one process each—even distributed across a network.

A dedicated *Relay* D-Bus object on the network is responsible for module administration and update notifications. At connection time, modules register with the relay, providing a list of IU categories and/or module names they are interested in. Category interests create loose functional links while module interests produce more static ones. Whenever a module chooses to publish information, it places a new IU in its right buffer, while removal of an IU from the right buffer corresponds to retraction. The relay is notified of such changes and in turn invokes a notification callback in all interested modules synchronising their left buffers by immediately and transparently creating or removing proxies of those IUs.

IUs consist of the fields described in the abstract model, and an additional category field which the relay can use to identify the set of interested modules to notify. They furthermore feature an optional custom lifetime, on the expiration of which they are automatically retracted.

Incremental changes to IUs are simply realised by changing their attributes: regardless of their location in either a right or left buffer, the same setter functions apply (e.g., `set_payload`). These generate relay-transported update messages which communicate the ID of the changed IU. Received update messages concerning self-owned and remotely-owned objects are discerned automatically to allow for special treatment of own IUs. The complete process is illustrated in Figure 1.

Current state and discussion. Our support for bi-directional IU editing is an extension to the concepts of the general model. It allows higher-level modules with a better knowledge of context to revise uncertain information offered by lower levels. Information can flow both ways, bottom-up and top-down, thus allowing for diagnostic and causal networks linked through category interests.

Coming from the field of embodied conversational agents, and being especially interested in modelling human-like communication, for example for on-line production of listener backchannel feedback, we constantly have to take incrementally changing uncertain input into account. Using the presented framework consistently as a network communication layer, we are currently modelling an entire cognitive architecture for virtual agents, based on the principle of incremental processing.

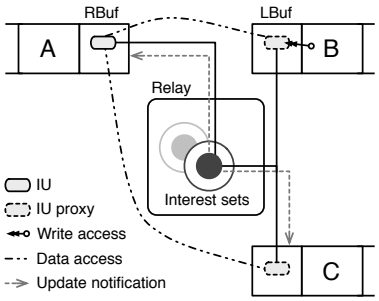

Figure 1: Data access on the IU proxies is transparently delegated over the D-Bus; module A has published an IU. B and C are registered in the corresponding interest set, thus receiving a proxy of this IU in their left buffer. When B changes the IU, A and C receive update notifications.

The decision for D-Bus as the transportation layer has enabled us to quickly develop versions for Python, C++ and Java, and produced straightforward-to-use libraries for the creation of IU-exchanging modules: the simplest fully-fledged module might only consist of a periodically invoked main loop callback function and any subset of the four handlers for IU events (added, removed, updated, committed).

4 Inpro Toolkit

The InproTK developed at UP offers flexibility on how tightly or loosely modules are coupled in a system. It provides mechanisms for sending IU updates between processes via a messaging protocol (we have used OAA [Cheyer and Martin, 2001], but other communication layers could also be used) as well as for using shared memory within one (Java) process. InproTK follows an event-based model, where modules create events, for which other modules can register as Listeners. Module networks are configured via a system configuration file which specifies which modules *listen* to which.

Modules *push* information to their right, hence the interface for inter-module communication is called *PushBuffer*. (At the moment, InproTK only implements left-to-right IU flow.) The `PushBuffer` interface defines a *hypothesis-change* method which a module will call for all its listening modules. A hypothesis change is (redundantly) characterised by passing both the complete current buffer state (a list of *IU*s) as well as the *delta* between

the previous and the current state, leaving listening modules a choice of how to implement their internal update.

Modules can be fully event-driven, only triggered into action by being notified of a hypothesis change, or they can run persistently, in order to create endogenous events like time-outs. Event-driven modules can run concurrently in separate threads or can be called sequentially by a push buffer (which may seem to run counter the spirit of incremental processing, but can be advantageous for very quick computations for which the overhead of creating threads should be avoided).

IUs are typed objects, where the base class `IU` specifies the links (same-level, grounded-in) that allow to create the IU network and handles the assignment of unique IDs. The payload and additional properties of an IU are specified for the IU's *type*. A design principle here is to make all relevant information available, while avoiding replication. For instance, an IU holding a bit of semantic representation can query which interval of input data it is based on, where this information is retrieved from the appropriate IUs by automatically following the grounded-in links. IU networks ground out in `BaseData`, which contains user-side input such as speech from the microphone, derived ASR feature vectors, camera feeds from a webcam, derived gaze information, etc., in several streams that can be accessed based on their timing information.

Besides IU communication as described in the abstract model, the toolkit also provides a separate communication track along which *signals*, which are any kind of information that is not seen as incremental hypotheses about a larger whole but as information about a single current event, can be passed between modules. This communication track also follows the observer/listener model, where processors define interfaces that listeners can implement.

Finally, InproTK also comes with an extensive set of monitoring and profiling modules which can be linked into the module network at any point and allow to stream data to disk or to visualise it online through a viewing tool (ANON 2009), as well as different ways to simulate input (e.g., typed or read from a file) for bulk testing.

Current state and discussion. InproTK is currently used in our development of an incremental multimodal conversational system. It is usable in its current state, but still evolves. We have built and integrated modules for various tasks (post-processing of ASR output, symbolic and statistical natural language understanding [ANON 2009a,b,c]). The configuration system and the availability of monitoring and visualisation tools enables us to quickly test different setups and compare different implementations of the same tasks.

5 Jindigo

Jindigo is a Java-based framework for implementing and experimenting with incremental dialogue systems currently being developed at KTH. In Jindigo, all modules run as separate threads within a single Java process (although the modules themselves may of course communicate with external processes). Similarly to InproTK, IUs are modelled as typed objects. The modules in the system are also typed objects, but buffers are not. Instead, a buffer can be regarded as a set of IUs that are connected by (typed) same-level links. Since all modules have access to the same memory space, they can follow the same-level links to examine (and possibly alter) the buffer. Update messages between modules are relayed based on a system specification that defines which types of update messages from a specific module go where. Since the modules run asynchronously, update messages do not directly invoke methods in other modules, but are put on the input queues of the receiving modules. The update messages are then processed by each module in their own thread.

Jindigo implements a model for updating buffers that is slightly different than the two previous approaches. In this approach, IUs are connected by *predecessor* links, which gives each IU (words, widest spanning phrases from the parser, communicative acts, etc), a position in a (chronologically) ordered stream. Positional information is reified by super-imposing a network of position nodes over the IU network, with the IUs being associated with edges in that network. These positional nodes then give us names for certain update stages, and so revisions can be efficiently encoded by reference to these nodes. An example can make this clearer. Figure 2 shows five update steps in the right buffer of an incremental ASR module. By reference to positional nodes, we can communicate easily (a) what the newest committed IU is (indicated in the figure as a shaded node) and (b) what the newest non-revoked or active IU is (i.e., the 'right edge' (RE); indicated in the figure as a node with a dashed line). So, the change between the state at time t_1 and t_2 is signalled by RE taking on a different value. This

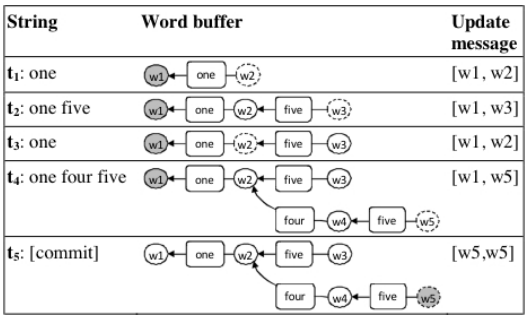

Figure 2: The right buffer of an ASR module, and update messages at different time-steps.

value (w3) has not been seen before, and so the consuming module can infer that the network has been extended; it can find out which IUs have been added by going back from the new RE to the last previously seen position (in this case, w2). At t_3, a retraction of a hypothesis is signalled by a return to a previous state, w2. All consuming modules have to do now is to return to an internal state linked to this previous input state. Commitment is represented similarly through a pointer to the rightmost committed node; in the figure, that is for example w5 at t_5.

Since information about whether an IU has been revoked or committed is not stored in the IU itself, all IUs can (if desirable) be defined as immutable objects. This way, the pitfalls of having asynchronous processes altering and accessing the state of the IUs may be avoided (while, however, more new IUs have to be created, as compared to altering old ones). Note also that this model supports parallel hypotheses as well, in which case the positional network would turn into a lattice.

The framework supports different types of update messages and buffers. For example, a parser may incrementally send NPs to a reference resolution (RR) module that has access to a domain model, in order to prune the chart. Thus, information may go both left-to-right and right-to-left. In the buffer between these modules, the order between the NPs that are to be annotated is not important and there is no point in revoking such IUs (since they do not affect the RR module's state).

Current state and discussion. Jindigo uses concepts from (Skantze, 2007), but has been rebuilt from ground up to support incrementality. A range of modules for ASR, semantic interpretation, TTS, monitoring, etc., have been implemented within the framework, allowing us to do experiments with complete systems interacting with users. We are currently using the framework to implement a model of incremental speech production.

6 Discussion

The three implementations of the abstract IU model presented above show that concrete requirements and application areas result in different design decisions and focal points.

While BF's approach is loosely coupled and handles exchange of IUs via shared objects and a mediating module, KTH's implementation is rather closely coupled and publishes IUs through a single buffer that lies in shared memory. UP's approach is somewhat in between: it abstracts away from the transportation layer and enables message passing-based communication as well as shared memory transparently through one interface.

The differences in the underlying module communication infrastructure affect the way incremental IU updates are handled in the systems. In BF's framework modules holding an IU in one of their buffers just get notified when one of the IU's fields changed. Conversely, KTH's IUs are immutable and new information always results in new IUs being published and a change to the graph representation of the buffer—but this allows an efficient coupling of module states and cheap revoke operations. Again, UP's implementation lies in the middle. Here both the whole new state and the delta between the old and new buffer is communicated, which leads to flexibility in how consumers can be implemented, but also potentially to some communication overhead.

In future work, we will explore if further generalisations can be extracted from the different implementations presented here. For now, we hope that the reference architectures presented here can already be an inspiration for further work on incremental conversational systems.

References

Adam Cheyer and David Martin. 2001. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March.

H. Pennington, A. Carlsson, and A. Larsson. 2007. *D-Bus Specification Version 0.12*. http://dbus.freedesktop.org/doc/dbus-specification.html.

David Schlangen and Gabriel Skantze. 2009. A General, Abstract Model of Incremental Dialogue Processing. In *Proceedings of EACL 2009*, Athens, Greece.

Gabriel Skantze. 2007. *Error Handling in Spoken Dialogue Systems*. Ph.D. thesis, KTH, Stockholm, Sweden, November.